Exploiting Prolific Types for Memory Management and Optimizations

Yefim Shuf [‡] [§] Manish Gupta [‡] Rajesh Bordawekar [‡] Jaswinder Pal Singh [§]

[‡] IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598
{yefim, mgupta, bordaw}@us.ibm.com

[§] Computer Science Department
Princeton University
Princeton, NJ 08544
{yshuf, jps}@cs.princeton.edu

ABSTRACT

In this paper, we introduce the notion of *prolific* and *non-prolific* types, based on the number of instantiated objects of those types. We demonstrate that distinguishing between these types enables a new class of techniques for memory management and data locality, and facilitates the deployment of known techniques. Specifically, we first present a new *type-based* approach to garbage collection that has similar attributes but lower cost than generational collection. Then we describe the *short type pointer* technique for reducing memory requirements of objects (data) used by the program. We also discuss techniques to facilitate the recycling of prolific objects and to simplify object co-allocation decisions.

We evaluate the first two techniques on a standard set of Java benchmarks (SPECjvm98 and SPECjbb2000). An implementation of the type-based collector in the Jalapeño VM shows improved pause times, elimination of unnecessary write barriers, and reduction in garbage collection time (compared to the analogous generational collector) by up to 15%. A study to evaluate the benefits of the short-type pointer technique shows a potential reduction in the heap space requirements of programs by up to 16%.

1. INTRODUCTION

A number of software and hardware technological trends point to the growing importance of automatic memory management and optimizations oriented towards reducing the cost of memory accesses. On the hardware side, the gap between processor and memory speeds continues to grow, motivating the need for optimizations to enhance data locality, including those that reduce the amount of memory being consumed by applications. On the software side, the use of object-oriented programming and reliance on automatic memory management is becoming more prevalent due to the accompanying productivity gains. In particular, the popularity of the Java programming language [21] has increased a great deal the interest in automatic memory management. Java is being widely used on systems ranging from embedded devices to high-end servers. On all of these systems, the efficient utilization of memory and reduced space requirements of applications (besides being inherently useful for memory-limited embedded devices) lead to higher performance and lower power consumption.

POPL '02, Jan. 16-18, 2002 Portland, OR USA
 ISBN 1-58113-450-9/02/01 ...$5.00.

Prolific and non-prolific types. In this paper, we introduce the notion of *prolific* and *non-prolific* types as a framework for improving automatic memory management. We present results from an empirical study of some well-known Java applications, which show that for each program, relatively few object types usually account for a large percentage of objects (and heap space) cumulatively allocated by the program. We refer to those frequently instantiated object types as *prolific* types and the remaining object types as *non-prolific* types. We suggest several optimizations that can potentially exploit the distinction between prolific and non-prolific types, both to improve performance with new memory management techniques as well as to simplify the deployment of some well-known techniques. In this paper, we primarily focus on two specific applications of the idea: type-based garbage collection [1] and reducing the amount of memory consumed by objects.

Type-based garbage collection We first present a novel *type-based* approach to garbage collection based on the notion of *prolific* and *non-prolific* object types. We propose a new *prolific hypothesis*, which states that objects of prolific types die younger than objects of non-prolific types. Our approach relies on finding garbage primarily among prolific objects. It is, therefore, conceptually similar to generational garbage collection, but it distinguishes between "generations" of objects based on type rather than age.

Generational garbage collection [29, 6] is one of the popular approaches to garbage collection. It is inspired by an observation, known as the *weak generational hypothesis*, that most objects die young [45]. A simple generational scheme involves partitioning the heap space into two regions – a *nursery* (or new generation) and an *old generation* (or a mature space) [2]. All new objects are allocated in the nursery. Most collections, termed *minor collections*, only reclaim garbage from the nursery. Survivors from a minor collection are promoted to the older generation, which is subjected to collection only during infrequent, *major collections*. In order to support a generational collection, the compiler has to insert a *write barrier* for each statement writing into a pointer field of an object, to keep track of all pointers from objects in the old generation to objects in the nursery. These source objects in the old generation are added as *roots* for minor collection, so that objects in the nursery which are reachable from those objects are not collected by mistake. Compared with their non-generational counterparts, generational collectors typically cause shorter pauses for garbage collection, due to the need to look at a smaller heap partition at a time, but lead to

[1] Not to be confused with *type-accurate* garbage collection.
[2] There are also multi-generational garbage collection schemes with more than two generations.

lower throughput of applications due to the overhead of executing write barriers.

In our type-based garbage collector, all objects of prolific types are assigned at allocation time to a *prolific region* (P-region), which is analogous to a *nursery* in a conventional generational collector. All "minor" collections are performed in the P-region. All objects of non-prolific types are allocated to a *non-prolific region* (NP-region), which corresponds to the *old generation* in a generational collector with two generations.[3] Unlike generational collection, objects are not "promoted" from the P-region to the NP-region after a minor collection. This approach leads to several benefits over generational collection:

- It allows a compiler to identify and eliminate unnecessary *write barriers* using simple type checks. This leads to performance benefits like:
  - reduction in the direct overhead of executing write barriers; and
  - for some write barrier implementations, a reduction in the number of roots that are considered during minor collections, leading to fewer objects being scanned and potentially fewer collections.
- It reduces the number of reference fields examined during garbage collection by using static type information to infer the direction of pointers.
- It avoids the problems associated with premature promotion of young objects that are going to die soon anyway, such as executing more write barriers, dragging dead objects into the old generation, and requiring more major collections.
- In a copying collector, the overhead of copying objects of non-prolific types across generations is avoided.

With an implementation of the type-based (non-copying) collector in the Jalapeño VM [2], the number of dynamically executed write barriers is reduced by up to 74% for SPECjvm98 [37] and SPECjbb2000 [38] benchmarks, and we observe shorter pause times. The total garbage collection times are reduced by an average of 7.4% over all benchmark programs, with an improvement of up to 15.2% for `javac`.

Reducing memory consumed by objects We also use the concept of prolific types in a technique that reduces the space requirements of applications written in object-oriented languages such as Java and increases the data locality in those applications. The technique relies on three observations: (i) the number of prolific types in a program is usually small, (ii) the pointers to type descriptors (run-time objects with class information) in two different objects of the same type are identical, (iii) objects of a prolific type are usually quite small. The *short type pointer* technique eliminates much of the space overhead in the headers of prolific objects by eliminating the pointer to a type descriptor and replacing it with only a few bits. An initial study on a suite of Java benchmarks (SPECjvm98, SPECjbb200, and JavaGrande application suite) indicates potential memory savings of 9% to 16% using this approach. Using smaller heap space can lead to less frequent garbage collections as well as fewer main memory accesses and improved data locality during program execution.

Other optimizations We describe a technique for recycling prolific objects. This technique can streamline the memory management of commonly allocated objects. We also briefly discuss a new approach to object co-allocation which relies on the prolificacy of types to make object co-allocation decisions. By placing closely-related objects together, this technique can improve the data locality of applications.

Organization The rest of the paper is organized as follows. Section 2 describes prolific and non-prolific types and techniques to identify prolific types. Section 3 discusses our proposed approach to type-based garbage collection. In Section 4, we discuss the short type pointer scheme. In Section 5, we describe other optimizations like object recycling and object co-allocation that can benefit from exploiting the distinction between prolific and non-prolific objects. Section 6 describes an implementation of the simplest, non-copying version of the type-based collection approach in the Jalapeño VM, and presents our experimental results; it also presents empirical data demonstrating the potential of the short type pointer technique. Section 7 discusses related work. Section 8 presents conclusions and Section 9 presents ideas for future work.

2. PROLIFIC AND NON-PROLIFIC TYPES

It is well known that for most applications, a large percentage of the program execution time is spent in a relatively small section of code. This behavior is exploited by adaptive runtime compilers like the Hotspot compiler [26] and the Jalapeño adaptive optimization system [4], as they focus expensive optimizations on those "hot-spots". It is not surprising that a similar hot-spot behavior is exhibited by object-oriented programs with respect to the types of objects that are created in those programs. In a study of some well-known Java benchmarks (namely, SPECjvm98 and SPECjbb2000), we have confirmed this observation. For example, in the `jack` program, 13 types account for 95% of all allocated objects, which also occupy 95% of the heap space allocated by this program. The data for other applications we studied can be found in Table 6, Figure 2, and Figure 3.

We define the term *prolific* to refer to a type that has a sufficiently large number of instances. In other words, a type is prolific with respect to a program if the fraction of objects allocated by the program that are of this type exceeds a certain threshold.[4] All remaining types are referred to as *non-prolific*.

2.1 Identifying Prolific Types

We now discuss a few approaches that may be used to identify prolific types. These approaches vary in terms of their overhead and accuracy. However, a misclassification of types does not create a correctness problem.

The simplest method of identifying prolific types is to use *offline profiling*. In an offline run, a runtime system monitors memory allocation requests issued by an application and counts the number of objects of each type that are allocated on behalf of an application. When an application exits, the collected allocation profile is saved by a JVM into a file. During an actual run, the runtime system uses previously collected allocation profiles to perform various optimizations. Thus, no monitoring overhead is incurred during the production run of the application.

An *adaptive approach*, in contrast to the offline profiling approach, is more accurate and attempts to identify prolific types during the actual production run of the program. An obvious adaptive strategy would be to monitor each memory allocation in an application. To reduce the overhead of monitoring object allocations, sampling techniques, such as those presented in [1], can be used.

It is not clear whether a static compile-time analysis *alone* (without profiling) can be effective in identifying prolific types. In pro-

[3] We can extend our approach to be analogous to a generational collector with several generations by defining multiple levels of prolificacy of types.

[4] In our experiments, the threshold is set to 1% of the total number of objects created by an application.

grams where the number of objects created at run time depends on the input data, it may not even be possible. Therefore, profiling is a good choice for determining prolific types.

2.2 Checking a Variable for Prolific Type

Most of our optimizations that exploit the distinction between prolific and non-prolific types require a compile-time test for whether an object is of a prolific type. In an object-oriented language with polymorphism, such as Java, this test requires analyses like *class hierarchy analysis* [16] or *rapid type analysis* [5], similar to those needed for inlining or devirtualization of methods. Given a declared type T of an object o, the compiler checks for o being definitely prolific by checking that all children of T in the class hierarchy are prolific.

Dynamic class loading [21] is another feature of Java that forces us to do compile-time analysis more conservatively. Again, the problem is similar to the problem with inlining of virtual methods in the presence of dynamic class loading [17, 36]. Due to dynamic class loading, a program can load a new class that is non-prolific but subclasses a prolific class, unless the prolific class is declared `final`.

It is possible to use the techniques for inlining virtual methods in the presence of dynamic class loading, like *preexistence analysis* [17] and *extant analysis* [36], to improve the effectiveness of the test for a type being definitely prolific. For example, using extant analysis, if we create a specialized method in which the reference to o is known to be extant (i.e., pointing to an existing object) [36], the test for o being prolific can be performed based on the existing class hierarchy, without worrying about any new classes that might be loaded.

We propose an alternate approach, described below, which leads to a much simpler compile-time test. We postulate that the prolific types are likely to be leaves, or close to leaves, in a type hierarchy. The intermediate classes are typically used for defining functionality that is common to all of their children classes. The subclasses refine the behavior of their parent classes and are usually instantiated more frequently than their respective parent classes. While it is possible that a prolific class may have one or more subclasses that are not prolific, we have made a choice to treat all children of prolific types as prolific. This greatly simplifies the test to check if a type is definitely prolific. The test returns `true` if the declared type of the variable is prolific, and returns `false` otherwise (without looking any further at the class hierarchy).

Our decision to treat the children of a prolific type as prolific seems to work well in practice. We have profiled all SPECjvm98 applications and the SPECjbb2000 benchmark and discovered that with three exceptions, prolific types are indeed the leaves in a type hierarchy. There are only two cases in which a subclass of a prolific class would have been regarded as non-prolific, but the artificial restriction we put in causes those subclasses to be treated as prolific.

3. TYPE-BASED MEMORY MANAGEMENT

In this section, we use the concept of prolific and non-prolific types to propose a *prolific hypothesis*. We then discuss a *type-based* approach to memory management, based on the prolific hypothesis.

3.1 The Prolific Hypothesis

We postulate a hypothesis that objects of prolific types have short lifetimes – we refer to it as the *prolific hypothesis*. An intuitive basis for this hypothesis is that if this were not true, an application that continuously allocates dynamic memory would have unsustainable memory requirements, as it would keep creating objects of prolific types at a fast pace without reclaiming sufficient space. Stated another way, our hypothesis predicts that the objects of prolific types die younger than objects of non-prolific types. It follows that most of the garbage collectible at various stages of the application would consist of objects of prolific types.

We validated this hypothesis and found that the relative survival rates are usually lower for objects of prolific types than for objects of non-prolific types. We also found that most of the dead objects and most of the garbage comes from short-lived objects of prolific-types. The empirical data can be found in [32].

Interestingly, the prolific hypothesis has some resemblance to a phenomenon commonly found in nature. Offsprings of prolific species are often short-lived [47].

3.2 Type-Based Approach

Our approach is to distinguish between prolific and non-prolific objects in the heap and direct the collection effort first towards prolific objects.

3.2.1 Type-based allocation

The *type-based* memory allocator partitions heap space into a *prolific region* and a *non-prolific region*: P-region and NP-region, respectively. The actual allocation mechanism is related to the kind of collector used by the system. When used with a copying collector, the allocator uses different regions of memory for the P-region and NP-region. With a non-copying collector, the objects of prolific and non-prolific types are tagged differently, but not necessarily allocated in separate memory regions.

When an object of a certain type is to be allocated, the allocator checks the type profile of the application (with information about whether or not the type is prolific) to decide whether to place the object in the P-region or NP-region. Hence, compared to a traditional memory allocator, the allocation path of the type-based allocator would have an extra step for checking the type of the object. However, since the prolificacy of types is known at compile-time, the compiler can avoid the overhead of the run-time type check by simply inserting a call to (or inlining) a specialized version of the allocation routine for prolific or non-prolific types.[5]

3.2.2 Type-Based Collection

Based on the prolific hypothesis, the *type-based* garbage collector assumes that most objects of prolific types die young, and performs (frequent) "minor" collections only in the P-region. Since objects of prolific types account for most of heap space, we hope to collect enough garbage on each *P-region collection*. When a P-region collection does not yield a sufficient amount of free space, a *full collection* of both P-region and NP-region is performed. If enough unreachable objects are uncovered during a P-collection, full collections will be infrequent. Objects remain in their respective regions after both P-region and full collections – i.e., unlike generational collection, objects that survive a P-region (minor) collection stay there and are not "promoted" to the NP-region. This enables the compiler to eliminate unnecessary write barriers with a relatively simple type check, as described in Section *3.2.3*. Since the survival rates of objects in the P-region are usually low, we expect the "pollution" of the P-region due to longer lived objects to be insignificant.

To ensure that during a P-region (minor) collection no object reachable from an object in the NP-region is collected, we have to keep track of all pointers from objects in the NP-region to objects in the P-region. This is accomplished by executing a write barrier code for pointer assignments, which records such inter-region references and places them in a *write buffer*. The contents of the write buffer represents *roots* used in a P-region collection.

[5] Our implementation does not perform this optimization.

Table 1: The percentage of dynamic assignments into the reference fields of objects of prolific types

Benchmark	% of $P \rightarrow \{P, NP\}$
compress	53
db	1
jack	99
javac	42
jess	99
mpegaudio	63
mtrt	80
jbb	73

3.2.3 *Eliminating Unnecessary Write Barriers with Compile-Time Analysis*

In the type-based collection, we do not move objects from the P-region to the NP-region or vice versa. Hence, unnecessary write barriers (other than those that keep track of references from the NP-region to the P-region) can be eliminated at compile time based on a simple type check. More specifically, given an assignment statement where the pointer of an object of type `source` is assigned a value corresponding to a pointer of an object of type `target`, we express the condition for eliminating a write barrier as:

```
EliminateWB  =  MustBeProlific(source)
            or  MustBeNonProlific(target)      (1)
```

Potential Opportunity. Table 1 shows the percentage of pointer assignments into the reference fields of objects of prolific types, i.e., those for which `IsProlific(source)` is true at run time. These data were obtained by running SPECjvm98 benchmarks (with size 100) and the SPECjbb2000 benchmark with the Jalapeño VM, using the optimizing compiler and a non-copying generational collector. The high percentages for all programs, except for db, show that there is clearly a potential to eliminate a substantial percentage of write barriers. Note that the numbers presented in Table 1 only give an estimate regarding how many of the write barriers can be eliminated (based on the `IsProlific(source)` part of the test). The actual numbers may be lower due to language features like polymorphism and dynamic class loading that introduce conservativeness in the compiler analysis.

Dealing with Polymorphism and Dynamic Class Loading We use the approach described in Section 2.2 of ensuring that each subclass of a prolific type is (artificially) regarded as prolific. This leads to a simple compile-time test for eliminating write barriers, which can be applied without worrying about any new classes that may be dynamically loaded in the future. The test described in (1) above is simplified to:

```
EliminateWB  =  IsDeclaredProlific(source)
            or  MustBeNonProlific(target)      (2)
```

3.2.4 *Processing Fewer Pointers*

In the type-based scheme, the number of pointers processed during a P-region collection can be reduced: not all pointers stored in objects that are scanned need to be examined.

During the P-region scanning process, for each object, the garbage collector requests the list of reference fields. This list is created when a class is loaded. Normally, the returned list contains all such reference fields. Consequently, all such references are first processed and then some of them (e.g., in a generational scheme, those that point to young objects) are scanned. However, in the type-based scheme, there is no need to return a complete list of reference fields to the collector during a P-region collection. Only the references pointing to objects of prolific types have to be returned (because object residing in the NP-region are only scanned during a full collection). To support this optimization, the class loader needs to provide to the collector with two different sets of methods returning the lists of reference fields: one (returning a partial list) for a P-region collection and one (returning a full list) for a full collection. (Our current implementation does not perform this optimization yet. Therefore, the performance of our implementation can be improved further.)

Table 2: Many pointers scanned during garbage collection are reference fields in object headers that point to type information block (TIB) objects (i.e., type descriptors).

	References Scanned		
Benchmark	# of TIB refs.	# of all refs.	% of TIB refs.
compress	8885294	28923650	30.719
db	1561864	2795719	55.866
jack	1446534	3796136	38.105
javac	4563270	14301008	31.908
jess	1940900	6551758	29.624
mpegaudio	409520	1421784	28.803
mtrt	2139610	3873905	55.231
jbb	2008508	5582408	35.979

3.2.5 *Avoiding Processing References to Type Descriptors*

We will now discuss a special case of the optimization technique presented in Section *3.2.4*. Usually, one of the fields in an object header points to a special object describing its type (or class) information: a type descriptor. For example, in the Jalapeño VM, this field points to a type information block (TIB) and is called a TIB field. Table 2 provides data showing that a large fraction of scanned pointers (28%-55% depending on the benchmark) are TIB pointers. Scanning TIB pointers for every reachable object is not necessary and can be avoided in the type-based scheme.

It is sufficient for only one object of a type to survive a collection to ensure that the TIB of that object is scanned and marked as live. The scanning of TIB fields in all other instances of that type is unnecessary, although the garbage collector will realize after reaching the TIB object that it has already been marked.

Since the number of distinct types is small, the number of TIB objects representing them is also small. It follows that such objects can be classified as instances of a non-prolific type and placed in the NP-region. As a result, the TIB fields (which now point to the NP-region) do not have to be examined during a P-region collection.

3.3 Discussion

The type-based approach, while similar to the generational approach in spirit, has some important differences.

- It involves pre-tenuring objects of non-prolific types into the heap partition which is collected less often. These objects do not need to be scanned during P-region (minor) collections. However, we expect those savings to be limited because non-prolific types, by their very nature, would not have occupied a lot of space in the nursery.

- Objects of prolific types are never promoted to the heap partition which is collected less often. This can be a double-edged sword. If objects of prolific types live for a long time, they can pollute the P-region, causing the scanning time to increase during future minor collections.[6] However, this ap-

[6] Our experimental results show that the times for minor collections

proach can also help avoid the negative side effects of premature promotion of young objects which are going to die soon anyway (namely, executing more write barriers; dragging more objects via write buffers into the old generation; and requiring more major collections).

- The separation between the heap partitions is based on static characteristics, i.e. the types of objects, rather than dynamic characteristics such as their ages. This allows unnecessary write barriers to be eliminated with a simple (and well-known in the context of dealing with virtual methods) compile-time analysis. This, apart from saving the direct overhead of executing write barriers, can also help avoid adding unnecessary objects to the write buffer, thus leading to fewer roots for minor collections, and potentially, more effective garbage collection.

- During a P-region collection, only objects of prolific types have to be scanned. As a result, reference fields that can only point to objects of non-prolific types do not even need to be examined. Fewer pointers to be examined translates into shorter garbage collection pauses. This optimization is also a consequence of the type-based nature of the division. It is possible because in our scheme, the assignment of an object to a separate region of collection depends only on the prolificacy of its type. This optimization cannot be performed in a generational scheme in which a reference field can point to either the nursery or the mature space; because the direction of a pointer cannot be known without actually examining it, all reference fields have to be checked for pointers into the nursery.

The performance implications of these characteristics will be explored in Section 6.

4. USING SHORT TYPE POINTERS TO REDUCE HEAP SPACE REQUIREMENTS

Each object in Java has an object header whose fields contain (or refer to) various bookkeeping information used by the Java Virtual Machine (JVM) and its components such as a garbage collector. A typical object header occupies two machine words. For example, in the Jalapeño VM, one of the fields in the object header, the status field, is used to support garbage collection and synchronization. The other field, the type field, is a class pointer and points to a special object describing the type of the object in question. Since most objects in Java programs are small (16-32 bytes),[7] the eight-byte object header carries 25%-50% space overhead. In this section, we use the notion of *prolific types* to describe a *short type pointer* technique which in many cases allows us eliminate the pointer to a type descriptor completely and reduce the length of the object header for many objects drastically (e.g., in half, for two-word object headers).

4.1 Exploiting Prolific Types

Our technique takes advantage of three observations and reduces the space requirements of Java applications. (The quantitative data will be presented in Section 6.2.) First, objects of the same type have the same pointer to a type descriptor. Second, only a handful of object types (prolific types) generate most of the objects that occupy most of the heap space. Third, objects of prolific types are usually small. We will now show how to eliminate the pointer to a type descriptor completely in many cases (for objects of prolific types) by utilizing only a few bits in the status field, which are usually available. Hence, the name – *a short type pointer* (STP).

A few bits, *type bits*, in the status field may be used to encode the types of prolific objects. A special value, say all zeros, is used to denote a non-prolific type. Given a maximum of k prolific types, we need $log_2(k+1)$ bits for this encoding (e.g., 4 bits for $k = 15$). The JVM creates a *type table* with an entry for the class object for each prolific type. Prolific objects no longer need a separate type pointer in the object header.

In order to get the type descriptor of an object (for a virtual method call or for an operation requiring a dynamic type check), its type bits are examined. If they do not contain the special value denoting a non-prolific type, the type bits are used to determine an index into the *type table*, which yields the needed type descriptor. Otherwise, the type descriptor is obtained via the type pointer which is stored as usual in the object header for non-prolific objects.

4.2 Discussion

The discussed technique has a number of important advantages. First, because no space is wasted for the pointer to the type descriptor in the object header of commonly occurring (prolific) objects, memory requirements of applications will be reduced noticeably. Smaller memory footprint can lead to higher performance due to better cache and page locality, and for long-running applications, less frequent garbage collections.

Although some extra instructions have to be executed to determine the type of an object, this overhead should be extremely small on modern superscalar processors. One of the extra instructions introduced is a memory (load) instruction. However, since it loads a reference from a very small type table which should fit in a few cache lines, most of those memory loads will hit in the first level cache. Furthermore, the frequency with which programs access the pointer to a type descriptor (for virtual method calls and dynamic type checks) is usually quite small in optimized Java codes [35].

The short type pointer approach has several advantages over the big bag of pages (BiBoP) technique [48] according to which objects of the same type are placed into the same page. While the BiBoP technique avoids the need to store the type information in any object, it has several disadvantages. First, since only objects of a particular type can reside in one page, objects of different types which point to each other cannot be allocated close to each other. Hence, the BiBoP approach may reduce the data locality of applications. Second, the BiBoP approach leads to the problem of memory fragmentation, as memory pages allocated for objects of many (non-prolific) types remain unfilled. Hence, this approach increases the TLB miss rates in applications that employ many different types of objects.

Interestingly, the notion of prolific and non-prolific types can be used to alleviate the memory fragmentation problem of the BiBoP technique by applying the BiBoP approach selectively and only to objects of prolific types. Consequently, all non-prolific objects would be co-allocated together. This would also reduce the TLB miss rates in applications with many different object types.

5. OTHER APPLICATIONS

In this section, we discuss additional applications of the notion of prolific and non-prolific types.

5.1 Object Recycling

Recycling dead objects is a technique that has often been used explicitly by programmers. Recently, Brown and Hendren [11]

are, in fact, lower for our type-based collector than for a generational collector.

[7] We verified this by profiling more than a dozen of Java programs from three different industry standard application suites. The data on object sizes is presented later in this paper.

Table 3: The reduction of write barrier overhead.

	Static count	Dynamic count					
	% of WB eliminated	# of write barriers executed			# of ref. added to the write buffer		
Benchmark	optimized	original	optimized	% elim.	original	optimized	% elim.
compress	0.000	3514	3514	0	149	149	0
db	3.048	81440517	81407032	0	63	54	14.2
jack	0.840	29326975	20849265	28.9	3912	3775	3.5
javac	31.967	41026080	33728942	17.8	714812	603352	15.5
jess	15.838	30777556	8106063	73.7	1765	1602	10.2
mpegaudio	0.000	17436480	17436480	0	310	310	0
mtrt	17.187	9832002	3513591	64.3	1073	167	84.4
jbb	0.622	26740265	19967459	25.3	46707	8803	71.1

Table 4: Comparison of the best throughput results (collected in the GC timing run). Execution times are given in secs. for SPECjvm98 programs (smaller is better) and throughput is given in ops/sec. for SPECjbb2000 (larger is better).

	Non-Copying GC				
	Generational	Non-Generational		Type-Based	
Benchmark	Raw	Raw	Normalized w.r.t Generational GC (%)	Raw	Normalized w.r.t. Generational GC (%)
compress	58.598	55.272	106.017	56.994	102.814
db	88.546	87.979	100.644	88.249	100.336
jack	51.619	52.478	98.363	50.732	101.748
javac	53.369	47.349	112.714	52.171	102.296
jess	31.728	34.813	91.138	31.122	101.947
mpegaudio	27.121	26.974	100.544	26.785	101.254
mtrt	22.485	23.239	96.755	22.975	97.867
jbb	1137.590	1119.240	98.386	1171.350	102.967

have discussed automating this technique using sophisticated whole-program interprocedural flow analysis. We propose a simpler approach that may capture many of the benefits.

As discussed earlier, objects of prolific types tend to have short lifetimes and account for most of the collectible garbage during program execution. Since most of the objects allocated in a program are of prolific types, it is worthwhile to recycle dead prolific objects by placing them into a special free pool instead of returning them to a general free pool. A special free pool is simply a linked list of free, not necessarily contiguous, objects of a particular type. Picking an object from such a pool requires a much shorter instruction sequence than allocating an object of arbitrary size. Hence, recycling of prolific objects can lead to more efficient allocation, especially with support for specialization of the allocation site based on the object type.

5.2 Object Co-allocation

Object co-allocation is known to have a positive impact on the performance of programs with heap allocated data [13, 14]. The problem is that it is usually very difficult to decide which objects should be co-allocated together. In this section, we will discuss a simple technique in which the knowledge of which types are prolific and which are non-prolific is used to drive object co-allocation decisions.

Considering our classification of objects into instances of prolific and non-prolific types, we argue that objects of prolific types should be allocated together with other objects of prolific types. First, in looking for instances of objects which are likely to be accessed together (such as objects connected via reference fields), there is a greater chance of a *one-to-one* relationship between objects of prolific types than between prolific and non-prolific objects (since there are many more prolific objects than non-prolific objects). Second, given the larger number of objects of prolific types, greater performance benefit may be achieved by focusing the effort on co-allocating prolific objects. We are investigating the effectiveness of this approach further [34].

6. EVALUATION

In this section, we provide an evaluation of our type-based memory management approach, based on an implementation in the Jalapeño VM [2]. We also provide a preliminary evaluation of the potential memory reduction from the STP technique, ahead of an actual implementation. We studied applications from the industry standard SPECjvm98 benchmark suite [37] and the SPECjbb2000 benchmark [38]. The SPECjbb2000 benchmark, which we refer to as `jbb`, is based on the pBOB (portable business object benchmark) [8], which follows the business model of the TPC-C benchmark [43] and measures the scalability and throughput of the JVM. We used the largest data set size (set to 100) to run SPECjvm98 applications.

6.1 Type-Based Memory Management

We now describe our implementation in the Jalapeño VM and the experimental results.

6.1.1 Implementation

The Jalapeño VM supports a number of different garbage collectors in different configurations of the VM. We implemented our type-based scheme by making several modifications to the non-copying generational garbage collector, which we found the simplest to start with.

Execution Modes. We modified the Jalapeño VM to introduce two modes of execution. In the profiling mode, the allocation profile is collected. In the production mode, the allocation profile collected in the profiling run is used by the memory allocator and garbage collector to implement type-based heap management, and by the compiler to optimize away unnecessary write barrier code.

Table 5: Garbage collection statistics. Execution times are given in seconds.

Non-copying GC Type-Based										
	# of GCs			GC Time			P-region col. pause time		Full col. pause time	
Benchmark	P-region	Full	Total	P-region	Full	Total	Min	Max	Min	Max
compress	2	94	96	0.184	50.935	51.120	0.068	0.121	0.531	0.631
db	14	3	17	5.041	2.682	7.724	0.150	0.172	0.590	1.061
jack	64	9	73	8.682	5.724	14.406	0.101	0.455	0.592	0.674
javac	28	31	59	20.845	46.723	67.569	0.232	0.815	0.595	1.935
jess	56	11	67	7.006	7.667	14.674	0.111	0.208	0.639	0.769
mpegaudio	1	2	3	0.177	1.222	1.400	0.179	0.179	0.592	0.634
mtrt	30	5	35	8.419	4.378	12.798	0.126	0.674	0.877	1.170
jbb	10	5	15	3.497	4.911	8.409	0.266	0.572	0.658	1.344
Non-copying GC Generational										
	# of GCs			GC Time			Minor col. pause time		Major col. pause time	
Benchmark	Minor	Major	Total	Minor	Major	Total	Min	Max	Min	Max
compress	2	94	96	0.193	55.786	55.979	0.102	0.142	0.604	0.756
db	14	3	17	5.314	2.967	8.282	0.180	0.186	0.661	1.202
jack	64	9	73	9.180	6.228	15.408	0.130	0.490	0.661	0.743
javac	53	38	91	17.468	62.253	79.721	0.250	0.913	0.672	2.119
jess	57	11	68	7.417	8.280	15.697	0.137	0.234	0.717	0.843
mpegaudio	1	2	3	0.183	1.313	1.497	0.209	0.209	0.660	0.703
mtrt	27	5	32	8.728	4.223	12.952	0.169	0.766	0.660	1.043
jbb	10	5	15	3.751	5.373	9.125	0.304	0.641	0.726	1.600
Non-copying GC										
	# of GCs			GC Time			Minor col. pause time		Major col. pause time	
Benchmark	Minor	Major	Total	Minor	Major	Total	Min	Max	Min	Max
compress	.	96	96	.	45.563	45.563	.	.	0.436	0.505
db	.	14	14	.	10.710	10.710	.	.	0.448	0.826
jack	.	62	62	.	31.271	31.271	.	.	0.436	0.619
javac	.	58	58	.	46.948	46.948	.	.	0.436	1.080
jess	.	64	64	.	36.630	36.630	.	.	0.439	0.689
mpegaudio	.	3	3	.	1.398	1.398	.	.	0.435	0.488
mtrt	.	26	26	.	19.018	19.018	.	.	0.435	0.874
jbb	.	11	11	.	9.137	9.137	.	.	0.447	0.963

Allocator and Collector. In the profiling mode, the memory allocator monitors all allocation requests, collects a type profile, and produces a file with the profile information, including class hierarchy information. In the production mode, the memory allocator uses the previously collected type profile to make allocation decisions. Also, in the production mode, the garbage collector repeatedly collects space occupied by dead objects of prolific types (P-region collections). When only a small portion of memory is freed, it collects the entire heap (full collections).

Write Barriers. We have made modifications to the write barrier code to ensure that the write barriers work appropriately for our type-based approach. We also modified the Jalapeño optimizing compiler to eliminate unnecessary write barriers during the production run of a program. The compiler analysis to identify unnecessary write barriers is based on the simplified test shown in Equation (2).

6.1.2 Experimental Results

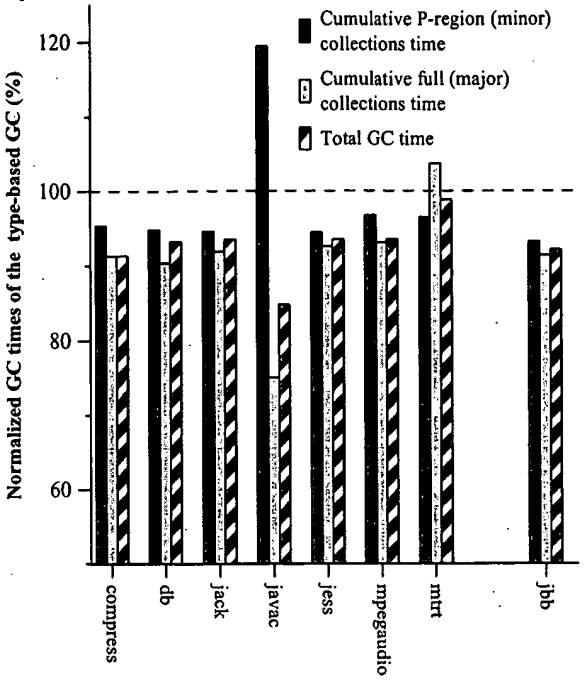
Our experiments were performed on an RS/6000 system with a 333 MHz PowerPC 604e processor and 768 MB of memory. For all SPECjvm98 benchmarks, we used a 64 MB heap, except for `javac` (80 MB). The `jbb` program ran with a 128 MB heap.

Reducing the Overhead of Write Barriers Table 3 demonstrates the effect of our write barrier elimination technique on reducing the overhead associated with execution of write barriers. Each SPECjvm98 program ran through three iterations. The `jbb` benchmark ran for two minutes after the initial ramp up stage.

For some benchmarks, the fraction of sites at which write barriers have been eliminated is fairly small (column 2). For others, it is quite significant and ranges from 15% to 32%. The number of write barriers eliminated at compile time does not translate linearly to the number of write barriers eliminated at run time (column 5). It can be seen that the programs that are amenable to our optimization execute 18%-74% fewer write barriers, which improves the throughput of these programs. Interestingly, a comparison with data presented in Table 1 suggests that there is a considerable potential for eliminating more write barriers by using more precise compiler analysis. In those programs, 3%-84% fewer entries are added to the write buffer for processing (column 8), which reduces the GC pauses and reduces the pollution of the heap. Benchmarks like `compress` and `mpegaudio` do not allocate a lot of objects which could be classified as instances of prolific types. Consequently, on these benchmarks, we are not able to eliminate write barriers. In `db`, most of the pointer assignments were to a few large arrays of references (which were classified as non-prolific) used for sorting data. As a result, the reduction of write barrier overhead for that benchmark is insignificant.

Performance and Throughput of Applications Table 4 shows the execution times of SPECjvm98 applications and the throughput of SPECjbb200 benchmark under the Jalapeño VM built with different garbage collectors (GC): our non-copying type-based GC, the non-copying generational GC, and the non-copying (non-generational mark-and-sweep) GC. Interestingly, our scheme yields the highest throughput numbers on three benchmarks, most

Figure 1: Normalized GC times of the type-based GC (with respect to the generational GC).



notably `jack` and `jbb`.

Compared to the generational GC, our scheme performs up to 3% better on all benchmarks, except for `mtrt` where the throughput is 2% worse. (We believe this anomaly is due to data locality effects and are investigating this further.) This is an encouraging result, suggesting that our scheme can improve the throughput of applications in systems where the generational GC is used.

Compared to the non-generational non-copying GC, our scheme performs noticeably better (by almost 12%) on `jess`, and somewhat better on `jack`, `mtrt` and `jbb` on which it shows a 1.5%-4.7% improvement. The non-generational GC performs better than our scheme on `compress`, `db`, and `mpegaudio`, probably because these programs do not generate a lot of objects. This program behavior limits the opportunities where our optimizations can apply.

Garbage Collection Statistics. Table 5 shows the benefits of our technique for garbage collection. The data was collected during performance runs for which the data was presented above. Compared to the non-copying generational collector, our scheme has fewer garbage collections during the execution of `javac` and `jess`. Interestingly, on `javac`, the number of both P-region and full collections (compared to minor and major collections in the generational scheme) is reduced. On `mtrt`, the number of P-region collections went up slightly. On all other benchmarks, the number of collections is the same. Overall, except for the "atypical" Java benchmarks like `compress` and `mpegaudio`, both the type-based and the generational collectors have a higher number of collections than the non-generational GC (which is expected). However, the number of expensive full collections is significantly smaller in the type-based scheme (and in the generational scheme) compared to the non-generational GC. This is done at the "expense" of performing more frequent, less-costly P-region (minor) collections.

Compared to the generational scheme, the type-based GC spends less time collecting the P-region. The `javac` program is an exception. For this benchmark, the average time spent on collecting the P-region is approximately twice as much as the time spent on collecting the nursery in the generational scheme. At the same time, the average time spent on collecting the whole heap is noticeably less (25%). Although the type based-scheme executes more inexpensive collections than the generational scheme during the execution of the `mtrt` benchmark, its average time for collecting young objects is smaller. With exception of `mtrt`, the total time spent on collecting the whole heap is smaller in the type-based scheme than in the generational GC. Finally, for all benchmarks, the total time spent on garbage collection is smaller in the type-based scheme compared to the generational GC. The improvements range from 1.2% to 15.2%, with an average improvement of 7.4%. Short average GC times is an attractive characteristic of the type-based scheme.

Both the minimum and maximum GC pauses during collection of young and all objects are shorter in the type-based scheme than those exhibited by the generational GC. This observation is important since short GC pauses is a critical requirement for some systems. The `mtrt` program is an exception where the maximum pause time for a full collection is slightly longer.

6.2 Short Type Pointers

In order to evaluate the potential benefits of the short type pointer (STP) technique before implementing this scheme, we instrumented the Jalapeño VM. We present the data we obtained, which serves as a preliminary evaluation. In addition to the SPECjvm98 and SPECjbb2000 benchmarks, we selected applications from the Java Grande suite for this study.

Table 6 presents various information about all objects as well as the number of types of prolific scalar and array objects. It can be seen that the number of prolific types is very small. Usually, there are less than 16 of them. This implies that we will need only 4 bits to encode interesting prolific types. Figures 2 and 3 illustrate that most of the objects are instances of prolific types and that instances of prolific types consume much of heap space, respectively.

Figure 4 depicts the average sizes of all objects as well as the sizes of prolific scalar and array objects. Object sizes include the sizes of object headers: 8 bytes for scalar and 12 bytes for array objects. It appears that prolific array objects are often twice bigger than prolific scalar objects. Interestingly, the average size of prolific scalar objects (PSO) in non-scientific applications ranges from 16 to 27 bytes while in scientific applications the sizes of PSO range from 30 to 39 bytes. This difference in sizes is partially due to a frequent use of double types in scientific programs and a large number of instance fields in some prolific objects. For objects that are 16-32 bytes long, one four-byte type pointer carries a considerable 12.5%-25% space overhead.

Table 7 shows how much space can be saved by shortening object headers in prolific objects. Eliminating the type pointer from the headers of prolific scalar objects (PSO) leads to 10%-25% reduction of space occupied by PSO. The same technique applied to the headers of prolific array objects (PAO), results in somewhat smaller (8%-16%) reduction of space occupied by PAO. Larger sizes of PAO contribute to this difference.

As can be seen from the data in Table 7, most of the savings come from shortening the headers of PSO. This is due to the fact that PSO consume much of the heap space but tend to be smaller than PAO. For some programs (e.g., `jack`, `jbb`, `search`), shortening headers of prolific arrays is also beneficial. Programs like `compress` and `montecarlo` which allocate huge arrays will not benefit from smaller object headers. Overall, 9%-16% of heap space can be saved by eliminating the type pointer from the headers of prolific objects (both arrays and scalars). Figure 5 illustrates the data presented in Table 7.

Table 6: Basic characteristics of SPECjvm98, SPECjbb2000, and Java Grande benchmarks.

	All Objects			# of Prolific Types	
Benchmark	# of Instances	Space	Aver. Size	Scalar	Arrays
compress	2318291	1241294165	535	12	2
db	35803659	879328007	24	3	1
jack	91557185	3398711330	37	7	6
javac	84072706	2602769995	30	9	2
jess	85505057	2938589866	34	5	2
mpegaudio	3571666	106567458	29	14	2
mtrt	65782126	1488212740	22	8	1
jbb	33319753	893288468	26	11	6
euler	16699233	650826152	38	2	-
montecarlo	2894017	1807520655	624	13	6
raytracer	66680479	219480438	30	2	-
search	35258805	1544528206	43	-	1

Table 7: Heap space reduction as a result of shortening object headers in prolific scalar objects and prolific array objects.

	Prolific scalar objects (PSO)		Prolific array objects (PAO)		Total space reduction (PSO + PAO)
Benchmark	% of PSO space	% of All space	% of PAO space	% of All space	% of All space
compress	17.87	0.48	9.53	0.06	0.54
db	24.52	14.40	10.86	0.77	15.17
jack	17.10	5.60	15.72	5.66	11.26
javac	20.25	6.08	8.91	2.87	8.95
jess	14.67	7.45	8.20	3.57	11.02
mpegaudio	16.98	8.93	9.46	1.15	10.08
mtrt	19.81	15.45	10.00	0.42	15.87
jbb	17.53	7.76	10.00	4.03	11.79
euler	10.17	9.41	-	-	9.41
montecarlo	12.81	0.32	0.21	0.20	0.52
raytracer	13.20	12.87	-	-	12.87
search	-	-	9.06	8.93	8.93

Figure 2: Most of the objects are instances of prolific types.

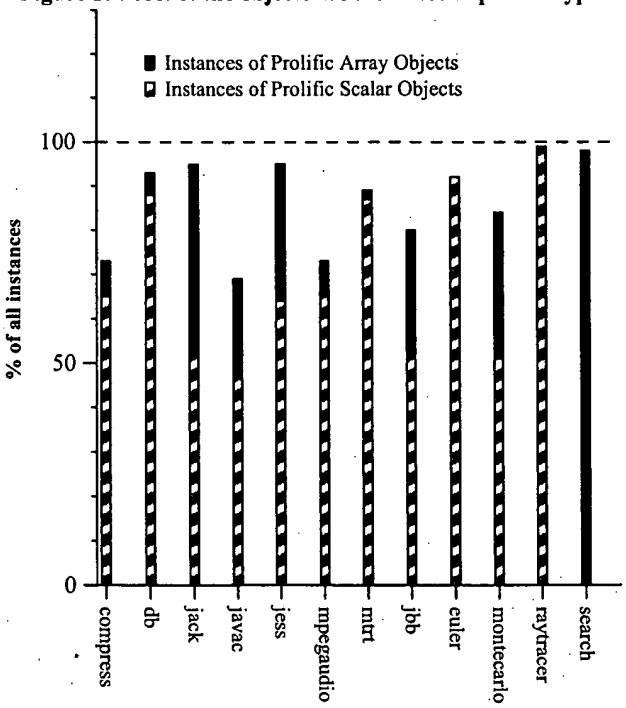


Figure 3: Most of the heap space is consumed by instances of prolific types.

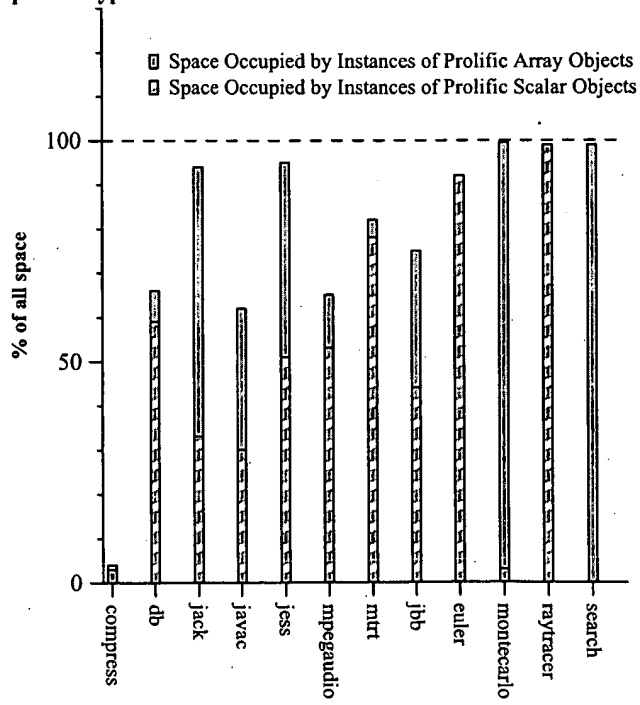


Figure 4: Average sizes of prolific and all objects.

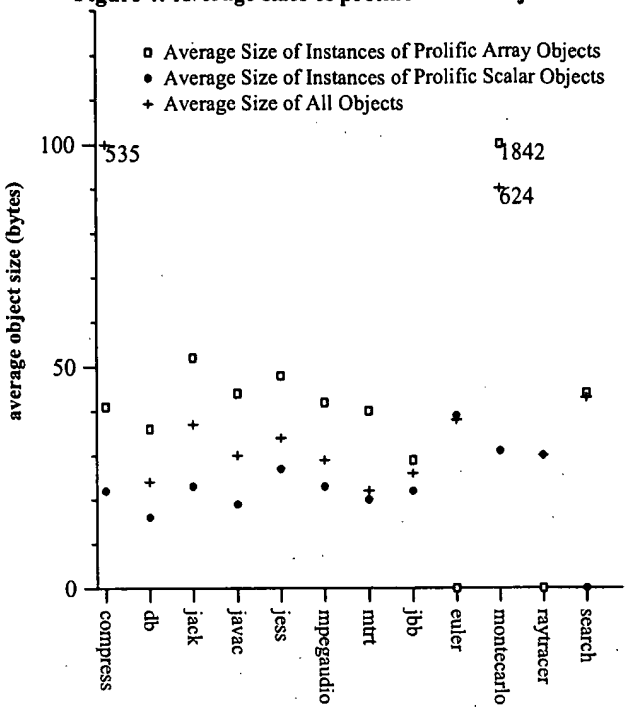
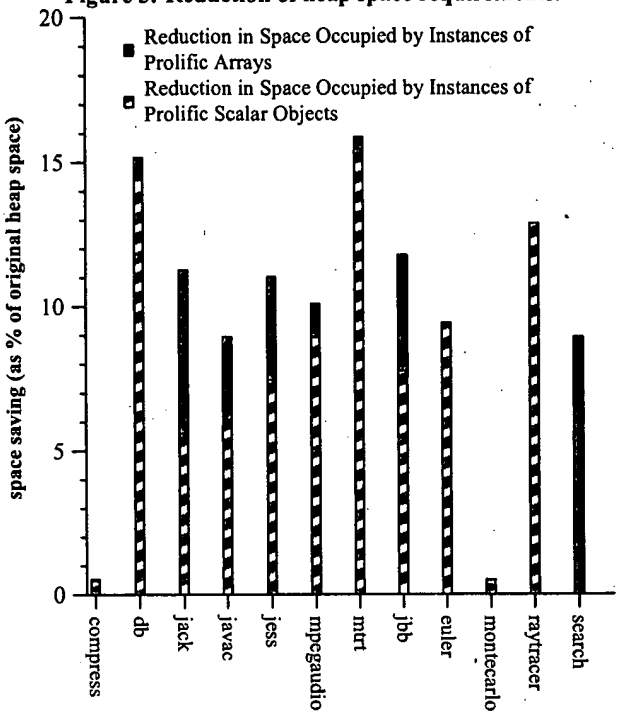


Figure 5: Reduction of heap space requirements.



7. RELATED WORK

The notion of "hot regions" in a program, wherein a program spends much of its execution time in small sections of code, is well-known. However, we are not aware of any previous work that applies the analogous idea in the context of different object types.

Garbage collection Jones and Lins [28] present a comprehensive overview of various memory allocation and garbage collection strategies. Surveys by Wilson [48], and Wilson, Johnstone, and others [49], discuss uniprocessor garbage collection and dynamic memory allocation algorithms, respectively.

Exploitation of object lifetimes for garbage collection has been investigated extensively. The key observation that the lifetime of many objects is short was reported as early as 1976 [18]. This insight then led to the formulation and exploitation of the *weak generational hypothesis* [45] which states that most objects die young. This hypothesis forms the basis of generational garbage collection [29, 6]: focus on reclaiming objects that are most likely to die, i.e., young objects.

Stefanovic et al. have investigated alternatives to the traditional *young-first* generational collectors [39]. They propose *age-based* garbage collection [40] algorithms, some of which use an *older-first* collector that collects older objects before the younger ones. This approach primarily reduces copying costs over the traditional generation collectors. Both the age-based and traditional generational collectors follow the same philosophy: both use age as a criterion for identifying objects for collection (young objects in generational and old objects in age-based collectors). On the contrary, our *type-based* garbage collector uses the prolificacy of object types as the criterion for identifying prospective moribund objects.

Experimental studies by Zorn [50], and Tarditi and Diwan [42] have shown that the cost of generational garbage collection is between 5% to 20%. Generational collectors usually segregate heap objects by their age. However, substantial performance improvement can be achieved by allocating large objects in a separate non-copy region, usually termed as *large object space*(LOS) [12]. Identification of large objects can be an absolute measure (e.g., more than 1024 bytes [46] or 256 bytes [27]) or a relative one (i.e., identify the object type whose instances occupy substantial space [25]). Many recent generation collection implementations use the LOS for storing large objects [23, 2].

The cost of write barriers is also significant, especially for pointer-intensive applications [42]. While there have been several efforts for improving the write barrier performance [24, 25], we did not come across any work that eliminates write barriers via static compile-time analysis.

Previous studies have investigated off-line feedback-driven approaches for segregating objects using criteria such as object lifetime and reference behavior. Barrett and Zorn [7] use full-run profiles on allocation-intensive C programs to predict short-lived objects, place them contiguously and delay their deallocation until large 4KB batches become free. Seidl and Zorn [31] propose partitioning heap for storing objects according to their reference behavior (the frequently vs. infrequently referenced objects) and lifetimes (short-lived vs. long-lived) objects. Blackburn et al. describe profile-driven technique for reducing copying by *pre-tenuring* long-lived objects, i.e., storing long-lived objects in an uncollected region [9]. Recently, Harris [22] has presented a dynamic technique in which selection of objects for pre-tenuring is performed at runtime.

Stefanovic et al. [41] describe analytical models for object lifetimes in object-oriented programs. Appel [3] has proposed that a plausible object lifetime distribution should use the following property: the expected future lifetime of an object is proportional to its current age.

Reducing space requirements We have compared the STP approach with the *big bag of pages* (BiBoP) technique in Section 4.2. The STP approach is different from the techniques aimed at producing compact code for embedded processors [30, 15], in that it reduces memory consumed by data rather than code. It is also different from the hardware-based techniques for compressing the contents of main memory [44].

A technique for delayed allocation of infrequently accessed or cold objects, a *construction on demand*, which aims to reduce space occupied by cold objects was suggested in [35].

Object recycling Bonwick [10] describes a slab allocator, a kernel memory allocator that reduces the cost of allocating complex objects by retaining the state of those objects between their uses. On the other hand, we consider object recycling of prolific objects in the context of a Java Virtual Machine.

Recently, Brown and Hendren [11] proposed an "object recycling mechanism" and implemented it as an automatic compiler optimization in the Soot Java optimization framework. Their compiler performs a conservative local program analysis to identify dead Java objects (and return them to per-class object pools) and transforms a program to recycle dead objects (from the object pools) instead of creating new ones.

Object co-allocation Object co-allocation at allocation time, based on hints supplied by the programmer, was reported by Chilimbi et al in [13]. Object co-allocation at garbage collection time was described in [14]. Object inlining was proposed by Dolby et al. in [19, 20].

8. CONCLUSIONS

We introduced a new framework of *prolific* and *non-prolific* types based on the number of instances of those types. This *type-based* rather than age-based framework serves as the foundation for several techniques that aim to improve memory management and data locality of programs with dynamic memory allocation.

We have presented a new *type-based* approach to garbage collection. Our approach directs the frequent collections towards objects of prolific types, much like generational collection directs them towards young objects. This leads to some important advantages over generational collection – fewer write barriers, potentially more effective collections, need to scan fewer pointers, and lower copying costs (the last one not verified yet, because our current implementation only performs non-copying collection).

With a preliminary implementation of this approach in the Jalapeño VM, we have observed significant improvements over the generational collector. For the SPECjvm98 and SPECjbb2000 benchmarks, the number of dynamically executed write barriers is reduced by 18% to 74% (except for three programs, for which there is no reduction). The total garbage collection times are reduced by an average of 7.4% over all benchmark programs. The overall performance improves modestly for most programs.

Based on the same framework, we have presented a technique for reducing the memory requirements of object-oriented applications. The technique is based on the observations that objects of the same type have the same type information pointer and that the number of prolific types is very small. The basic idea is to eliminate type pointers from the headers of prolific objects (which tend to be small and occupy much of the heap space) and to encode the types of prolific objects using only a few bits instead. The empirical data we collected shows that when applied to twelve Java programs (from SPECjvm98, SPECjbb200, and the Java Grande suites of applications) representing a variety of workloads, this technique can save 9%-16% of heap space.

Finally, we presented the use of prolific types for recycling and co-allocation of objects. The former approach is based on the observation that most of the dead objects are those of prolific types and attempts to recycle them instead of returning them to the main free pool. The latter approach is based on the idea that an object of a prolific type is likely to be referenced through another object of prolific type.

9. FUTURE WORK

This work opens up a number of interesting possibilities for future research. We plan to implement and measure the impact of the short type pointer technique on program performance and on the frequency of garbage collections. We also plan to investigate the impact of object recycling and object co-allocation techniques discussed in this paper on locality and performance [34].

Another interesting direction would be to investigate a more dynamic version of our automatic memory management approach, where the detection of prolific types is done during program execution in an adaptive manner. Changing the status of a type from prolific to non-prolific, or vice versa, would require selective recompilation of sections of code where write barriers may have been eliminated based on the older classification of types (we do not anticipate a need to undo the *effect* of a previously eliminated write barrier, as long as the existing objects are not moved across their respective regions).

It would also be interesting to develop a copying version of our type-based collector. This would also involve allocating objects of prolific and non-prolific types in distinct regions of memory, which will have a further impact on the data locality characteristics.

Designing and implementing a hybrid scheme, which combines the characteristics of both type-based and age-based approaches, is another interesting direction for future research. We are investigating the effectiveness of this approach further [33].

Acknowledgments

We would like to thank the members of the Jalapeño team for providing us with an infrastructure for this work. We would also like to thank Pratap Pattnaik for valuable technical discussions.

10. REFERENCES

[1] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. In *Proc. of ISMM 2000*, pages 127–136, 2000.

[2] B. Alpern, C. R. Attanasio, J. J. Burton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shephard, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. *IBM Systems Journal*, 39(1):194–211, 2000.

[3] A. Appel. A better analytical model for the strong generational hypothesis, November 1997.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proc. of OOPSLA 2000*.

[5] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of OOPSLA 1996*, pages 324–341, 1996.

[6] H. G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, 1993.

[7] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. *ACM SIGPLAN Notices*, 28(6):187–196, June 1993.

[8] S. J. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, 2000.

[9] S. Blackburn, J. Cavazos, S. Singhai, A. Khan, K. McKinley, and J. E. B. Moss. Profile-driven pretenuring for Java. Poster in OOPSLA 2000, October 2000.

[10] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference*, pages 87–98, 1994.

[11] R. Brown and L. Hendren. Automatic recycling of Java objects. Technical Report, McGill University, 2000.

[12] P. J. Caudill and A. Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *Proc. of OOPSLA 1986*, pages 119–130.

[13] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proc. of PLDI*, pages 1 – 12, 1999.

[14] T. M. Chilimbi and J. R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proc. of the 1998 ISMM*, Oct. 1998.

[15] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. of PLDI 1999*, pages 139 – 149, Atlanta, GA USA, May 1999.

[16] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy. In *Proc. of 9th European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[17] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proc. of 13th ECOOP*, pages 258–278, 1999.

[18] L. P. Deutch and D. Bobrow. An efficient incremental automatic garbage collector. *CACM*, 19(7), July 1976.

[19] J. Dolby. Automatic inline allocation of objects. In *Proc. of PLDI*, 1997.

[20] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *Proc. of PLDI 2000*, pages 345 – 357, 2000.

[21] J. Gosling, B. Joy, and G. Steele. *The Java(TM) Language Specification*. Addison-Wesley, 1996.

[22] T. Harris. Dynamic adaptive pre-tenuring. In *Proc. of ISMM'00*, pages 127–137, 2000.

[23] M. Hicks, L. Hornof, J. T. Moore, and S. M. Nettles. A study of large object spaces. In *Proc. of ISMM 1998*, pages 138–146.

[24] U. Hoelzle. A fast write barrier for generational garbage collectors. In *Proc. of OOPSLA'93 Workshop on Garbage Collection*, 1993.

[25] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *Proc. of OOPSLA 1992*, pages 92–109, 1992.

[26] The Java Hotspot performance engine architecture. `http://java.sun.com/products/hotspot/whitepaper.html`.

[27] R. Hudson, J. E. B. Moss, A. Diwan, and C. Weight. A language-independent garbage collector toolkit. Technical Report TR91-47, University of Massachusetts at Amherst, September 1991.

[28] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[29] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–429, June 1983.

[30] A. Rao and S. Pande. Storage assignment optimizations to generate compact and efficient code on embedded DSP. In *Proc. of PLDI 1999*, pages 128 – 138, Atlanta, GA USA, May 1999.

[31] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proc. of ASPLOS*, pages 12–23, 1998.

[32] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Distinguishing between prolific and non-prolific types for efficient memory management. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2001.

[33] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. A hybrid memory management scheme using prolific types and object ages. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2001.

[34] Y. Shuf, M. Gupta, H. Franke, and J. P. Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 2001.

[35] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proc. of SIGMETRICS 2001*.

[36] V. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proc. of PLDI 2000*, 2000.

[37] Standard Performance Evaluation Council. *SPEC JVM98 Benchmarks*, 1998. `http://www.spec.org/osg/jvm98/`.

[38] Standard Performance Evaluation Council. *SPEC JBB2000 Benchmark*, 2000. `http://www.spec.org/osg/jbb2000/`.

[39] D. Stefanović. *Properties of Age-based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, Amherst, MA, February 1999.

[40] D. Stefanović, K. McKinley, and J. E. B. Moss. Age-based garbage collection. In *Proc. of OOPSLA 1999*, pages 370–381, October 1999.

[41] D. Stefanovic, K. S. McKinley, and J. E. B. Moss. On models for object lifetime distributions. In *Proc. of ISMM 2000*.

[42] D. Tarditi and A. Diwan. The full cost of a generational copying garbage collection implementation. In *Proc. of the OOPSLA93 Workshop on Memory Management and Garbage Collection*, October 1993.

[43] Transaction Processing Performance Council. *TPC-C Benchmark*, 2000. `http://www.tpc.org/cspec.html` .

[44] R. Tremaine, P. Franaszek, J. Robinson, C. Schulz, T. Smith, M. Wazlowski, and P. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research Development*, 45(2), 2001.

[45] D. M. Ungar. Generational scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[46] D. M. Ungar and F. Jackson. Tenuring policies for generation based storage reclamation. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[47] A. R. Wallace. On the tendency of varieties to depart indefinitely from the original type. In Letters to the Royal Society, Feb. 1858.

[48] P. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas at Austin, 1994.

[49] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. of International Workshop on Memory Management*, September 1995.

[50] B. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, EECS Department, University of California at Berkeley, 1989.